Advanced Programming

Reexam: EddyCommand and Pubsubscriber

Kasper Passov pvx884

Contents

1 Eddy Parser

1.1 Parser choice

I chose to use the ReadP parser as i am most comfortable with it and i find the function munch very usefull, with the SimpleParser does not have.

1.1.1 Ident

To ensure there are at least one character in the ident i made a helper function contains String -¿ String -¿ Bool, that checks if the two lists has an element in common.

1.1.2 charN

charN allows any of the first 256 characters of the Unicode character set into the insert, and consumes characters until the length described by i is hit.

1.1.3 Integer

I made sure to return the integer as an integer and not a string, so it could easily be read by charN

1.1.4 commandTerm

i moved repeat to its own Nonterminal as describer by the grammar. the rest of the Terminals are unambiguous and without precedence and can as such be put into one Nonterminal.

1.2 Grammar

A welldefined grammar is not left-recursive and upholds operator precedence. I have made some leftfactorication to the given grammar to remove the left recursion of command. The only operatorions that calls for precedense is the commands macro and repeat, and the given grammar upholds the precendese.

1.2.1 Command

The command rule repeat has to be leftfactorized as repeat is left-recursive. I have removed the left recursion by changing the grammar of command to the following:

1.3 Testing

With an unambiguous grammar parsing scripts will be deterministic.

2 Eddy Interpreter

3 Pubsub

3.1 Implementation

I chose not to use OTP as i fell more comfortable building the server myself, where there are as little hiden code as possible. I realise this creates uglier and more code, but i generaly fell the quality of my implementation is improved by this choice.

3.1.1 Node

Ones startet, each node in the network has 3 empty lists of tuples. The first saves the nodes subs in tuples of process ids and filters [S, F]. This is done so the node can find and access each subscriber and the filter for that subscriber. The second list of tuples is a list of messages this node has recieved, and the uniqe id of that message [E, Ref]. The last list of tuples contains and the process that cast the error [E, S].

3.1.2 The Error list

The error list is populated when a filter fails on some input. This is firstof to uphold the robustness of the network so a filter throwing an error does not halt the network. Secondly it is saved so the users or adminstrators have some way of discovering why the nodes does not send the messages to the subscribers. For this i extended the API to contain an errors/1 function that takes a process id and returns that nodes list of filter errors.

3.1.3 Classes of Mistakes

The helper function tryFilter/4 is called when a node recieves a message and all subscribers Filter functions are to called on it. The helperfunction tries to run the filter on the element and returns the filters response if its returntype is a boolean. If the filter function throws some kind of error it is caught by the filters try/catch, the second type of bad filter i could think of, is one where the filter succeds, but does not return a boolean. In this case i have chosen to return false and inform the node of the filter not returning as expected. So the classes of mistakes the implementation can handle, is the filter throwing exceptions and type errors. One type of filter error i do not catch is that of a filter function that creates an infinite loop. In such a case the node will break as it does not know it is in an infinite loop and it waits for it to return.

3.1.4 Subscribtion loops

One potential weakness of the network is nodes subscribers creating loops between them. This could potentialy create infinite loops where a message is bounced between a number of subscribers when filters does not stop the message. The implementation of my network stops this by binding an uniqe reference to every message. Everytime a node recieves a message, it checks its message list of any references equal to that recieved. If one of those is found the message is not forwarded or saved. Assuming erlang:make_ref() (practicaly) creates an uniqe id, an already send message should never be saved or resend. This solution means it is possible to publish identical messages throughout the sytem without the messages being stopped, as all new messages have a new identifier. make_ref recycles ids after 2$\hat{8}$2 calls, meaning there are some posibility that a messages will be lost if enough messages is called. This number is large enough for it to be unique enough for practical purposes, if not in theory.

3.2 Testing

I use the lightweight testing framework EUnit to test my erlang implementation, and the analysing tool dialyzer. Dialyzer throws passes it with no warnings. I have made 8 manual tests for EUnit that can be run with pubsub:test(), this tests the general functionality of the API, and everything seams to work as expected. The demonstration asked for in the assignment can be run using pubsub:demonstration().

3.3 Evaluation

I belive this to be the best part of my solutions, blabla

4 Appendix

4.1 Appendix A: Parser

```
--
-- Skeleton for Eddy parser
-- To be used at the re-exam for Advanced Programming, B1-2013
--

----------------------------
-- Student: Kasper Passov --
-- KU ID:   pvx884        --
----------------------------

{-# OPTIONS_GHC -Wall -fno-warn-unused-do-bind #-}

module EddyParser
  ( Error
  , EddyParser.parse
  , parseFile
  , contains
  ) where

import Text.ParserCombinators.ReadP
import Data.Char
import EddyAst

data Error = Error deriving (Show, Eq)

-----------------------------
--           API          --
-----------------------------

parseFile :: FilePath -> IO (Either Error Script)
parseFile f = do s <- readFile f
                 return $ EddyParser.parse s

parse :: String -> Either Error Script
parse input = case output of
    (p, ""):_ -> Right p
    _         -> Left Error
  where output = readP_to_S (do
                  sc <- script
                  skipSpaces
                  eof
                  return sc) input
```

```
-----------------------------
-- Internal Implementation --
-----------------------------

script :: ReadP Script
script = commands

commands :: ReadP [Command]
commands = many command

command :: ReadP Command
command = do ct <- commandTerm
             rest ct
          where
             rest ct = edrepeat ct
                   +++ return ct

edrepeat :: Command -> ReadP Command
edrepeat ct = do skstring "*"
                 i <- integer
                 edrepeat $ Repeat i ct -- edrepeat is run agein if we have nested repeats
             +++ return ct

commandTerm :: ReadP Command
commandTerm = insert +++ del    +++ next  +++ prev +++
              buffer +++ remove +++ macro +++ call
            where
              insert = do skstring "i"
                          i <- integer
                          skstring "|"
                          cn <- charN i
                          return (Ins cn)
              del    = do skstring "del"
                          return Del
              next   = do skstring "next"
                          return Next
              prev   = do skstring "prev"
                          return Prev
              buffer = do skstring "buffer"
                          s <- ident
                          return (Buffer s)
              remove = do skstring "remove"
                          return Remove
              macro  = do skstring "macro"
                          st <- ident
```

```
                         skstring "{"
                         sc <- script
                         skstring "}"
                         return (Macro st sc)
             call   = do st <- ident
                         return (Call st)

integer :: ReadP Int
integer = do skipSpaces
             n <- many1 (satisfy isDigit)
             return $ read n

charN :: Int -> ReadP String
charN i = do c <- many (satisfy isLatin1)
             if length c == i
                then return c
                else pfail

ident :: ReadP String
ident = do skipSpaces
           cs <- munch(`elem` allowed)
           if checkIdents cs && contains cs (['a'..'z']++['A'..'Z'])
              then return cs
              else pfail
        where allowed = ['a'..'z']++['A'..'Z']++['0'..'9']++"_./*?"
              checkIdents s = s `notElem` reservedKeywords

contains :: [Char] -> [Char] -> Bool
contains []     _ = False
contains (h:t) l = contains' h l || contains t l

contains' :: Char -> [Char] -> Bool
contains' _ []     = False
contains' e (h:t)  = if h == e
                       then True
                       else contains' e t

-----------------------------
--          Misc           --
-----------------------------

skstring :: String -> ReadP String
skstring st = skipSpaces >> string st

reservedKeywords :: [String]
reservedKeywords = ["i", "del", "next", "prev", "buffer", "remove", "macro"]
```

4.2 Appendix B: Interpreter

```haskell
--
-- Skeleton for Eddy interpreter
-- To be used at the re-exam for Advanced Programming, B1-2013
--
{-# OPTIONS_GHC -Wall #-}
module EddyInterp where

import EddyAst

type Line = String
type Buffer = ([Line], [Line])

data EddyState = EddyState { buffers :: [(BufferName, Buffer)],
                             activeBuffer :: BufferName,
                             macros :: [(MacroName, Script)] }
                 deriving(Show, Eq)

newtype EddyCommand a = EC { runEddy :: EddyState -> (a, EddyState) }


instance Monad EddyCommand where
  return a = EC $ \e -> (a, e)
  a >>= f  = EC $ \e ->
              let (j, en) = runEddy a e in
                runEddy (f j) en

emptyState :: EddyState
emptyState = EddyState { buffers = [("*scratch*", ([],[]))],
                         activeBuffer = "*scratch*",
                         macros = [] }


----------------------------------------
-- Functions to manipulate the states --
----------------------------------------

-- returns the state
getState :: EddyCommand EddyState
getState = EC $ \s -> (s, s)

-- sets the current state to ns
setState :: EddyState -> EddyCommand ()
setState ns = EC $ const ((), ns)
```

```haskell
-------------------------
-- Buffer manipulation --
-------------------------

-- returns the buffer with the name bn
getBuffer :: BufferName -> EddyCommand Buffer
getBuffer bn = do (EddyState nbufs _ _) <- getState
                  case lookup bn nbufs of
                       Nothing -> getBuffer "*scratch*"
                       Just b  -> return b

-- adds a buffer b to the bufferlist
addBuffer :: (BufferName, Buffer) -> EddyCommand()
addBuffer (bn,b) = do (EddyState bufs _ ma) <- getState
                      setState(EddyState (bufs++[(bn,b)]) bn ma)

-- removes a buffer unless it is *scratch*
removeBuffer :: EddyCommand()
removeBuffer = do (EddyState b ab ma) <- getState
                  let newb = filter(\(x,_) -> x /= ab) b
                  if ab == "*scratch*"
                    then raiseError
                    else setState(EddyState newb "*scratch*" ma)

-------------------------
-- Active manipulation --
-------------------------

-- returns the active buffer
getActive :: EddyCommand Buffer
getActive = do (EddyState _ ab _) <- getState
               getBuffer ab

-- sets the active buffer to the buffer with the name bn
changeActive :: BufferName -> EddyCommand()
changeActive bn = do (EddyState b _ ma) <- getState
                     case lookup bn b of
                       Nothing    -> addBuffer (bn, ([],[])::Buffer)
                       Just _     -> setState (EddyState b bn ma)

-- update the active buffer to b
updateActive :: Buffer -> EddyCommand()
updateActive b = do (EddyState bufs bn ma)  <- getState
                    newb <- return $ filter (\(x,_) -> x /= bn) bufs ++ [(bn, b)]
                    setState (EddyState newb bn ma)
```

```
------------------------
-- Macro manipulation --
------------------------

-- adds a macro (mn, s) to the current macros
addMacro :: MacroName -> Script -> EddyCommand()
addMacro mn s = do (EddyState b ab ma) <- getState
                   newma <- return $ filter (\(x,_) -> x /= mn) ma ++ [(mn,s)]
                   setState (EddyState b ab newma)

-- runs the macro mn
runMacro :: MacroName -> EddyCommand()
runMacro mn = do (EddyState _ _ ma) <- getState
                 case lookup mn ma of
                   Nothing -> raiseError
                   Just sc -> script sc


------------------------
--   Error function   --
------------------------

raiseError :: EddyCommand()
raiseError = do es <- getState
                setState es

edrepeat :: Int -> Command -> EddyCommand()
edrepeat 0 _ = do es <- getState -- det her skammer jeg mig over
                  setState es
edrepeat i c = do command c
                  edrepeat (i-1) c


------------------------
--  API Functionality --
------------------------

script :: Script -> EddyCommand()
script []     = do es <- getState
                   setState es
script (c:sc) = do command c
                   script sc

command :: Command -> EddyCommand ()
command c = case c of
              (Buffer bn)    -> changeActive bn
              (Remove)       -> removeBuffer
              (Macro mn s)   -> addMacro mn s
              (Call m)       -> runMacro m
              (Repeat i co)  -> edrepeat i co
              _              -> command' c

command' :: Command -> EddyCommand ()
command' c = do ab <- getActive
                newb <- case c of
                        (Ins l) -> insertLine l ab
                        (Del)   -> delLine ab
                        (Next)  -> nextLine ab
                        (Prev)  -> prevLine ab
                        _       -> return ab -- Some error
                updateActive newb


-----------------------
-- Line manipulation --
-----------------------

insertLine :: Line -> Buffer -> EddyCommand Buffer
insertLine l (lb, la) = return (l:lb, la)

delLine :: Buffer -> EddyCommand Buffer
delLine (lb, []) = return (lb, [])
delLine (lb, _:la) = return (lb, la)

nextLine :: Buffer -> EddyCommand Buffer
nextLine (lb, []) = return (lb, [])
nextLine (lb, lah:la) = return (lah:lb, la)

prevLine :: Buffer -> EddyCommand Buffer
prevLine ([], la) = return ([], la)
prevLine (lbh:lb, la) = return (lb, lbh:la)



runeddy :: Script -> EddyCommand Buffer
runeddy sc = do script sc
                getActive

data Error = Error deriving(Show, Eq)

runScript :: Script -> Either Error [(BufferName, String)]
runScript sc = case buflist of
```

```
            a -> Right a
            {- _ -> Left Error -} -- errors are not implementet
where
buflist = map (\(bufname, (lprev, lnext)) -> (bufname, concatMap (++"\n") ((reverse lp
(_,(EddyState buffs _ _)) = runEddy ( runeddy sc ) emptyState
```

4.3 Appendix C: Pubsubscriber

```
%%%-------------------------------------------------------------------
%%% @author Ken Friis Larsen <kflarsen@diku.dk>
%%% @copyright (C) 2014, Ken Friis Larsen
%%% @doc
%%% Skeleton for the re-exam for Advanced Programming, B1-2013
%%% Implementation of pubsub server
%%% @end
%%%-------------------------------------------------------------------
%%% Student name: Kasper Passov
%%% Student KU-id: pvx884
%%%-------------------------------------------------------------------
-include_lib("eunit/include/eunit.hrl").
-module(pubsub).
-export([start/0, add_subscriber/3, stop/1, subscribers/1, publish/2, demonstration/0]).



%%%-------------------------------------------------------------------
%%% API
%%%-------------------------------------------------------------------

start() ->
    {ok, spawn(fun() -> psnode([],[],[]) end)}.

add_subscriber(P, S, F) ->
    rpc(P, {S, F, add_sub}).

subscribers(P) ->
    rpc(P, get_subs).

publish(P, E) ->
    Ref = erlang:make_ref(),
    info(P, {{E, Ref}, publish}).

messages(P) ->
    rpc(P, messages).


%%%-------------------------------------------------------------------
%%% API extensions
%%%-------------------------------------------------------------------
% These API extensions have been made to make testing easier

% The stop function kills the given process P. It does not kill or
% message its subscribers
stop(P) ->
    rpc(P, stop).

% Returns all errors the given process has saved. An error happens
% when a filter fails or returns something that is not a bool
errors(P) ->
    rpc(P, errors).
%%%-------------------------------------------------------------------
%%% Internal Implementation
%%%-------------------------------------------------------------------

psnode(Subs, Messages, Errors) ->
    receive
        {From, stop} ->
            reply(From, {stopped, self()});
        {From, errors} -> % return all errors
            reply(From, Errors);
        {Error, S, error} -> % save an error
            psnode(Subs, Messages, Errors ++ [{Error, S}]);
        {From, {S, F, add_sub}} -> % add a subscriber
            case lookupSubs(S, Subs) of
                  none -> reply_ok(From),
                          psnode([{S,F}] ++ Subs, Messages, Errors);
                  _ -> reply_error(From),
                       psnode(Subs, Messages, Errors)
            end;
        {From, get_subs} -> % return subscribers
            reply(From, getAll(Subs)),
            psnode(Subs, Messages, Errors);
        {{E, Ref}, publish} -> % forward message
            case lists:keyfind(Ref, 2, Messages) of
                false -> PassedList = lists:filter(fun({S,F}) -> tryFilter(self(),S,F,E) en
                         lists:foreach(fun({S,_}) -> info(S, {{E, Ref}, publish}) end, Passe
                         psnode(Subs, [{E, Ref}] ++ Messages, Errors);
                _     -> psnode(Subs, Messages, Errors)
            end;
        {From, messages} -> % return all messages
            reply(From, getAll(Messages)),
            psnode(Subs, Messages, Errors);
        _ ->
            erlang:display(end_of_pubserver) % helps me find lost messages
    end.
%%%-------------------------------------------------------------------
%%% Helper Functions
%%%-------------------------------------------------------------------
```

```erlang
tryFilter(From, S, F, E) ->
    try F(E) of
        true -> true;
        false -> false; % function fails
        _ -> info(From, {filter_not_bool, S, error}),
             false %function does not return bool
    catch % Funktion fails on type
    _ : _ -> info(From, {filter_fails, S, error}),
                 false %"Bad filter function"
end.

lookupSubs(_, []) -> none;
lookupSubs(S, [{S, F} | _]) -> F;
lookupSubs(S, [_ | SUBS]) -> lookupSubs(S, SUBS).

getAll([]) -> [];
getAll([{A, _} | Rest]) -> getAll(Rest) ++ [A].

%%%-------------------------------------------------------------------
%%% Demonstration
%%%-------------------------------------------------------------------

% demonstration as describer on page 9 of the exam set
demonstration() ->
    {ok, Niels} = start(),
    {ok, Albert} = start(),
    {ok, Christiaan} = start(),
    {ok, Isaac} = start(),
    {ok, Joseph_Louis} = start(),
    {ok, Johannes} = start(),
    {ok, Euclid} = start(),
    add_subscriber(Albert, Niels, fun(_) -> true end),
    add_subscriber(Isaac, Albert, fun(X) -> hd(tl(X)) == 101 end), % 101 is the character 'e
    add_subscriber(Christiaan, Albert, fun(_) -> true end),
    add_subscriber(Euclid, Isaac, fun(X) -> X rem 2 == 0 end),
    add_subscriber(Johannes, Isaac, fun(_) -> true end),
    add_subscriber(Isaac, Joseph_Louis, fun(_) -> true end),
    publish(Euclid, 5),
    publish(Euclid, 4),
    publish(Euclid, point),
    publish(Isaac, "Hello"),
    publish(Christiaan, {tick, tock}),
    publish(Albert, emc2),
    timer:sleep(10),
    {messages(Niels),messages(Joseph_Louis)}.
```

```
%%%-------------------------------------------------------------------
%%% Testing
%%%-------------------------------------------------------------------

% tests start and stop
start_test() ->
    {ok, P} = start(),
    {stopped, P} = stop(P).

% tests publish and messages
publish_one_test() ->
    {ok, P} = start(),
    publish(P, "Winter"),
    publish(P, "is"),
    publish(P, "comming"),
    ["Winter","is","comming"] = messages(P).

% tests adding subscribers
add_sub_test() ->
    {ok, A} = start(),
    {ok, B} = start(),
    {ok, C} = start(),
    add_subscriber(A, B, fun(_) -> true end),
    add_subscriber(A, C, fun(_) -> true end),
    [B, C] = subscribers(A).

% tests publishing to subscribers
publish_subs_test() ->
    {ok, A} = start(),
    {ok, B} = start(),
    {ok, C} = start(),
    add_subscriber(A, B, fun(_) -> true end),
    add_subscriber(A, C, fun(_) -> true end),
    error = add_subscriber(A, C, fun(_) -> true end),
    publish(A, "A is greatest"),
    timer:sleep(10),
    "A is greatest" = messages(A),
    "A is greatest" = messages(B),
    "A is greatest" = messages(C).

%tests filters
publish_filter_test() ->
    {ok, A} = start(),
    {ok, B} = start(),
    {ok, C} = start(),
    add_subscriber(A, B, fun(X) -> X < 4 end),
    add_subscriber(A, C, fun(X) -> X > 2 end),
    publish(A, 2),
    publish(A, 3),
    publish(A, 4),
    timer:sleep(10),
    AM = messages(A),
    BM = messages(B),
    CM = messages(C),
    {[2,3,4],[2,3],[3,4]} = {AM,BM,CM}.

% tests error handling and rebustness
error_function_test() ->
    {ok, A} = start(),
    {ok, B} = start(),
    {ok, C} = start(),
    add_subscriber(A, C, fun(_) -> throw(poo) end),
    add_subscriber(A, B, fun(X) -> X end),
    publish(A, "zoo abes throw "),
    timer:sleep(10),
    AE = errors(A),
    [{filter_not_bool,B}, {filter_fails,C}] = AE.

% message is recieved through nested sub
nested_subs_test() ->
    {ok, A} = start(),
    {ok, B} = start(),
    {ok, C} = start(),
    add_subscriber(A, B, fun(_) -> true end),
    add_subscriber(B, C, fun(_) -> true end),
    publish(A, "we can go deeper"),
    timer:sleep(10),
    ["we can go deeper"] = messages(C).

% messages are not continuously send
sub_loop_test() ->
    {ok, A} = start(),
    ok = add_subscriber(A, A, fun(_) -> true end),
    publish(A, "recurring"),
    timer:sleep(20),
    ["recurring"] = messages(A).

%%%-------------------------------------------------------------------
%%% Communication primitives
%%%-------------------------------------------------------------------
```

```
%% synchronous communication

rpc(Pid, Request) ->
    info(Pid, {self(), Request}),
    receive
        {Pid, Response} ->
            Response
    end.

reply(From, Msg) ->
    From ! {self(), Msg}.

reply_ok(From) ->
    reply(From, ok).

reply_error(From) ->
    reply(From, error).

%% asynchronous communication

info(Pid, Msg) ->
    Pid ! Msg.
```